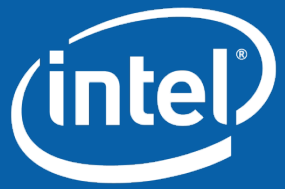


The Intel® Itanium® Architecture Basics

Cameron McNairy
Itanium® Processor Architect
Intel® Corporation

Agenda

Background

Application Architecture

- Explicit Parallel Instruction Computing, Compilers and Performance

System Architecture

- Versatile, Capable and Secure

Building Blocks and Background

Question: How to address the needs of the high end server market?

- Scalability, performance, adaptability, security, availability and resiliency

Answer: The Intel® Itanium® Processor Architecture

  - Application architecture emphasizes program performance, scalability, and security
  - System architecture emphasizes adaptability, scalability, availability, security and resiliency
- Application Architecture – great performance with opportunities to do better
  - Code generator identifies and forms parallelism (EPIC)
  - Simplified in order execution
  - Allow out of order memory hierarchy
- System Architecture – adaptable to multiple OSs, OEMs, and customers
  - Flexible, capable, large memory management and accessibility
  - Bi-endian, OS specific registers, efficient OS calling mechanisms
  - RAS+M for enterprise and mainframe with Machine Check Architecture

Application Architecture Features

Instruction-level parallelism

Large number of registers

- Register stack
- Software pipelining

Predication

- Remove hard branches
- Provide more ILP
- Software pipelining

Speculation

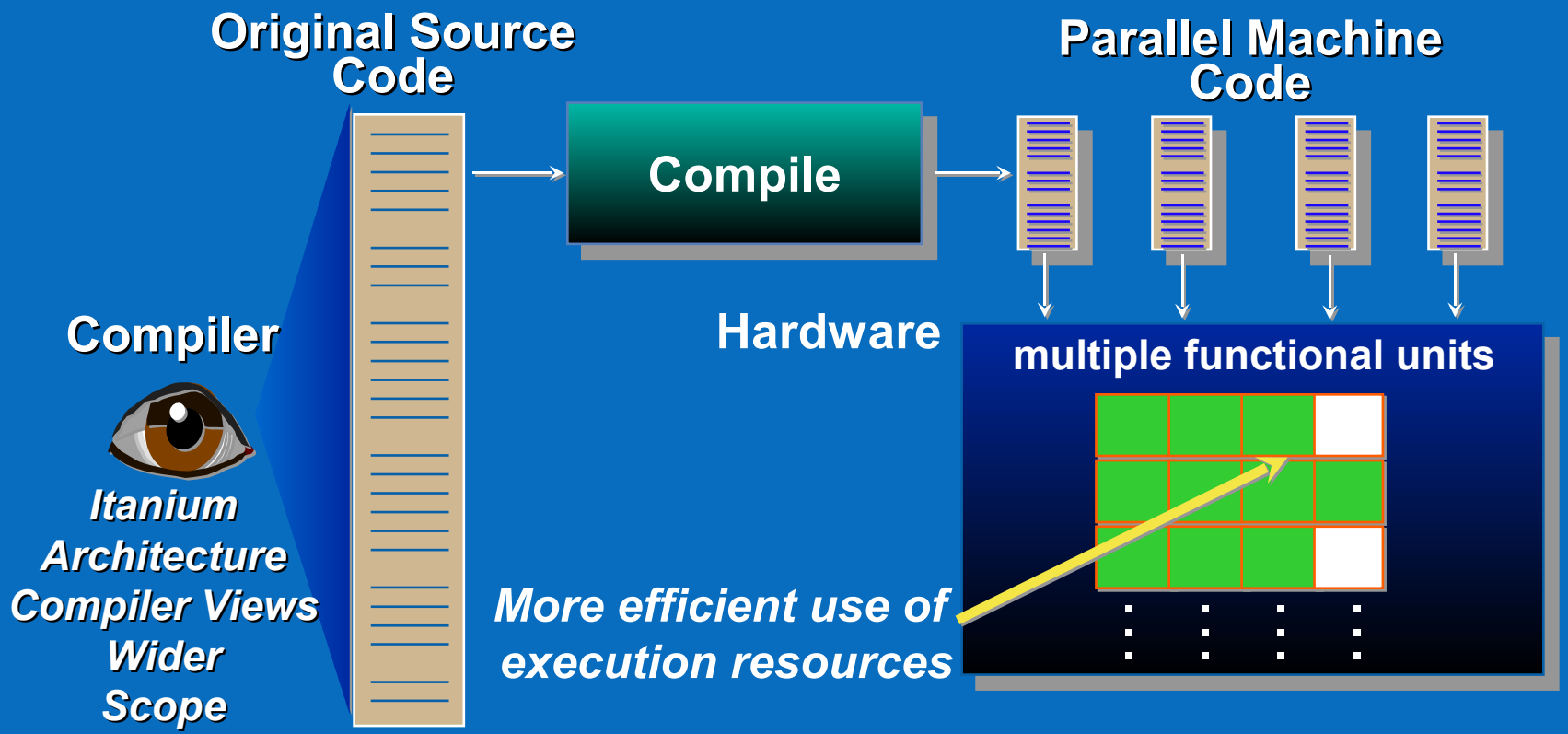
- Control
- Data

Hints

Floating Point

Memory

Identifying Instruction Level Parallelism



Instruction Group

An instruction group is one or more independent instructions that may be executed concurrently

Read after write (RAW) or write after write (WAW) dependencies block concurrence

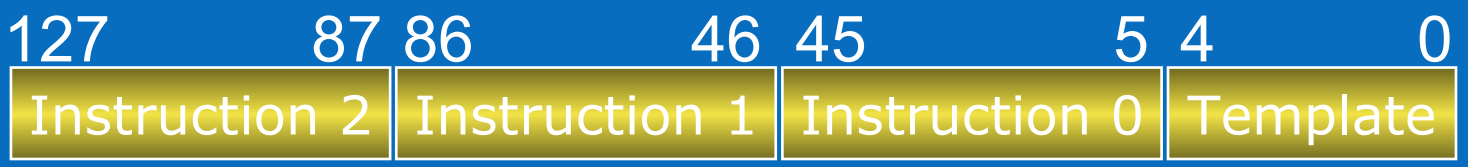
Instruction groups issue in parallel, depending on available resources

Instruction Bundle

Bundle: A group of two or three instructions (16 bytes total)

- Three instruction slots (41 bits each), a template field (5 bits)
  - Template breaks issue groups and defines instruction type

    A = ALU, I = Integer, M = Memory, F = Floating Point, B = branch, L = long
- Issue groups can span partial bundles or several bundles

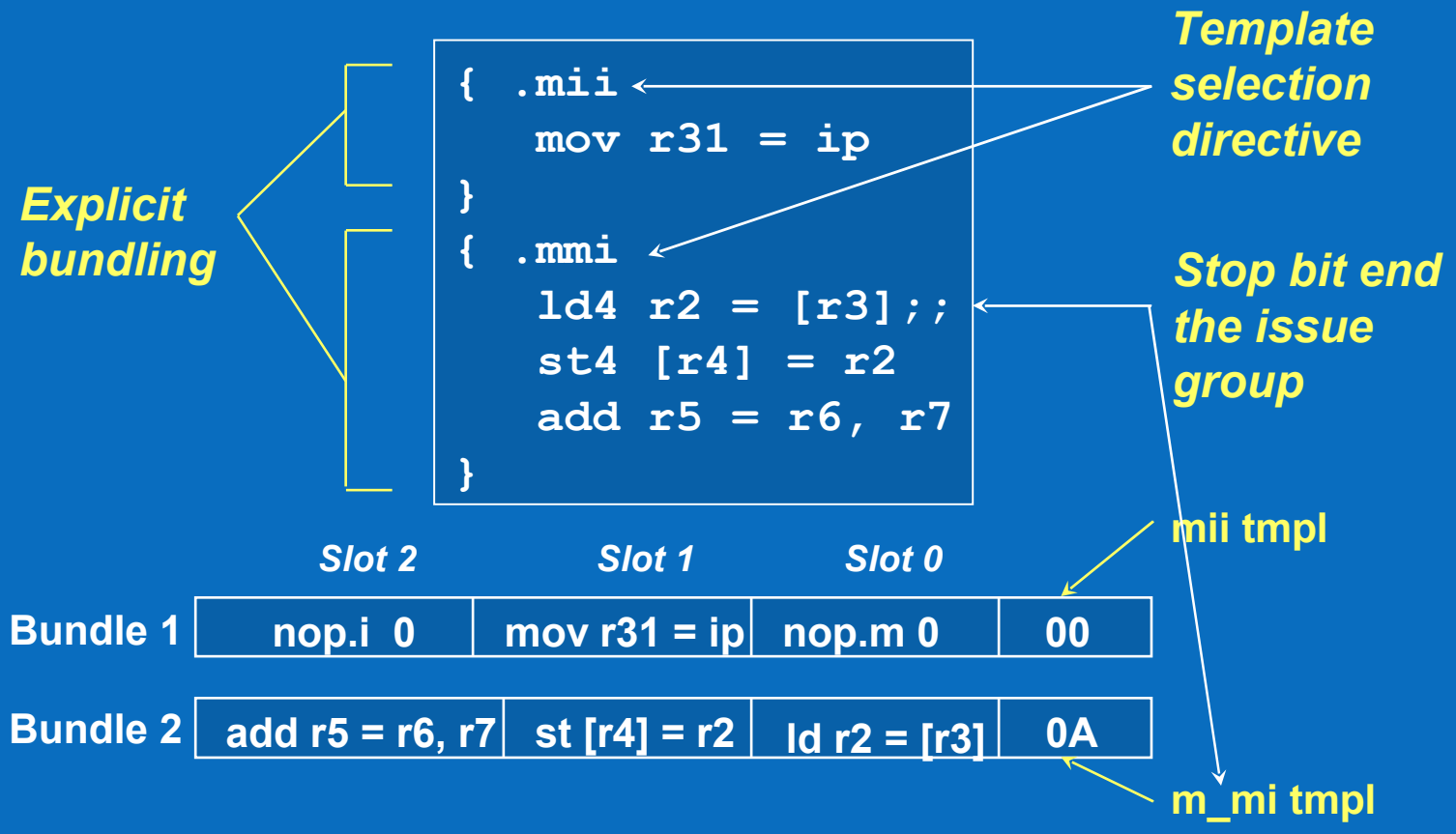


Instruction Templates

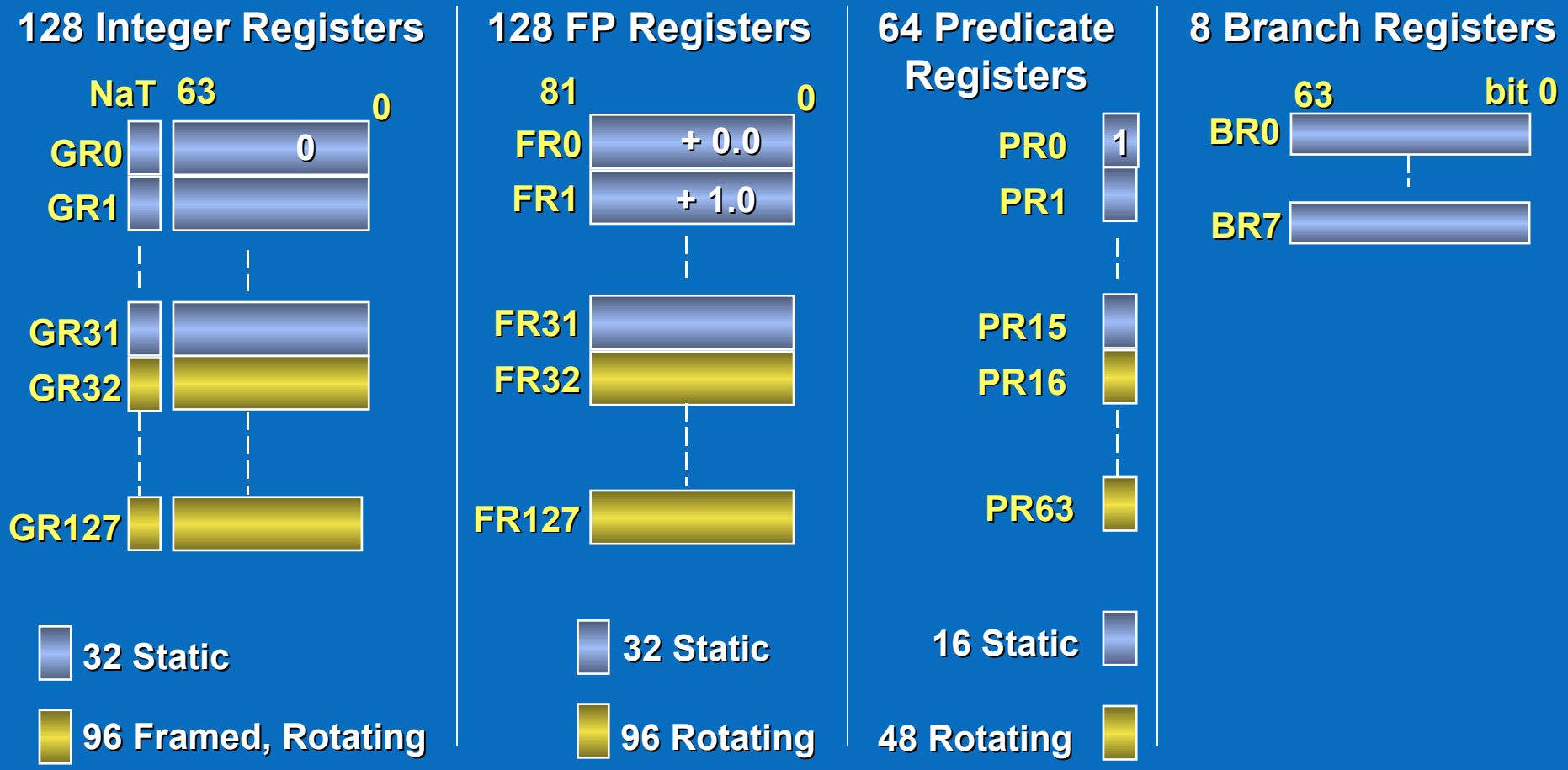
Slot 0	Slot 1	Slot 2
M	I	I
M	L	X
M	M	I
M	F	I
M	M	F
M	I	B
M	B	B
B	B	B
M	M	B
M	F	B

Instruction Type	Description	Execution Unit
A	Integer ALU	I or M
I	Non-ALU Integer	I
M	Memory	M
F	Floating Point	F
B	Branch	B
L/X	Extended	I or B

Assembly Language Example



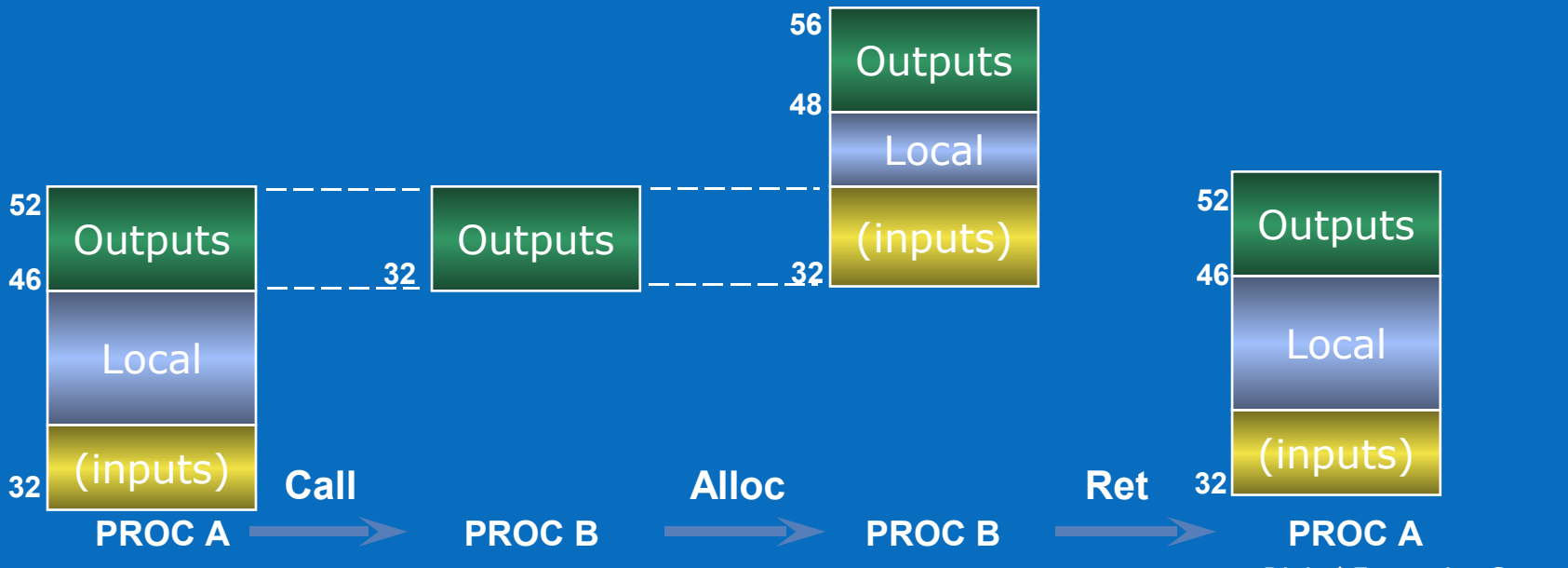
Large Register Set



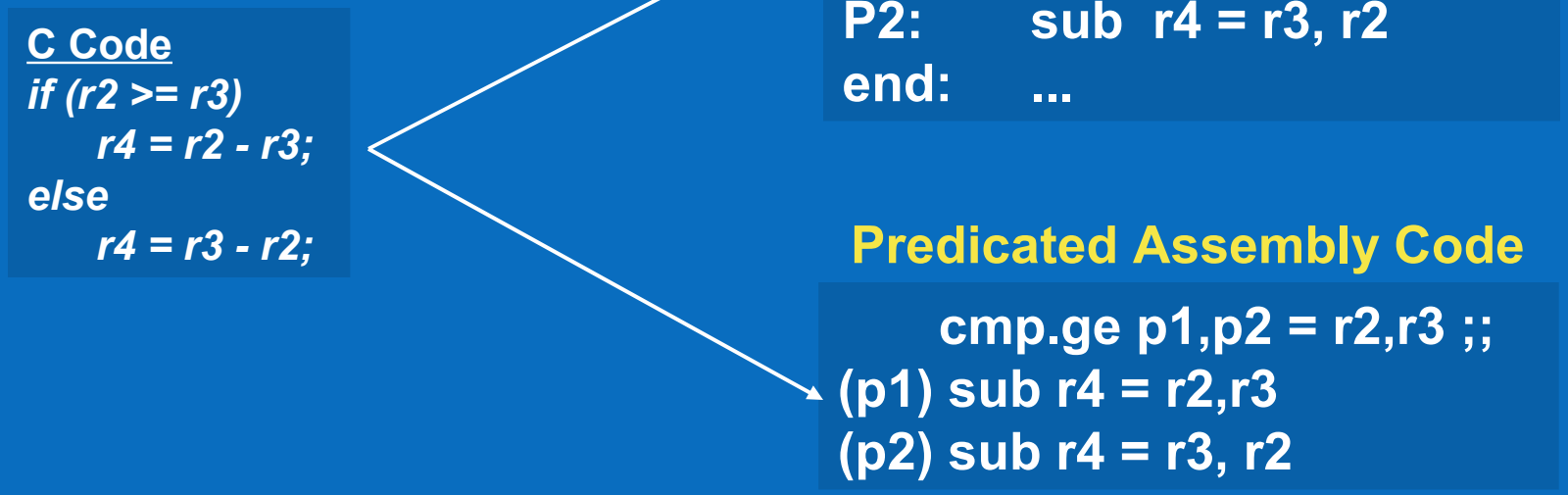
Register Stack

Virtually unlimited integer registers with the Register Stack Engine

- Alloc sets the frame to the desired size: local, output, and rotating
- Call changes frame to contain only the caller's output
- Return restores the stack frame of the caller



Predication



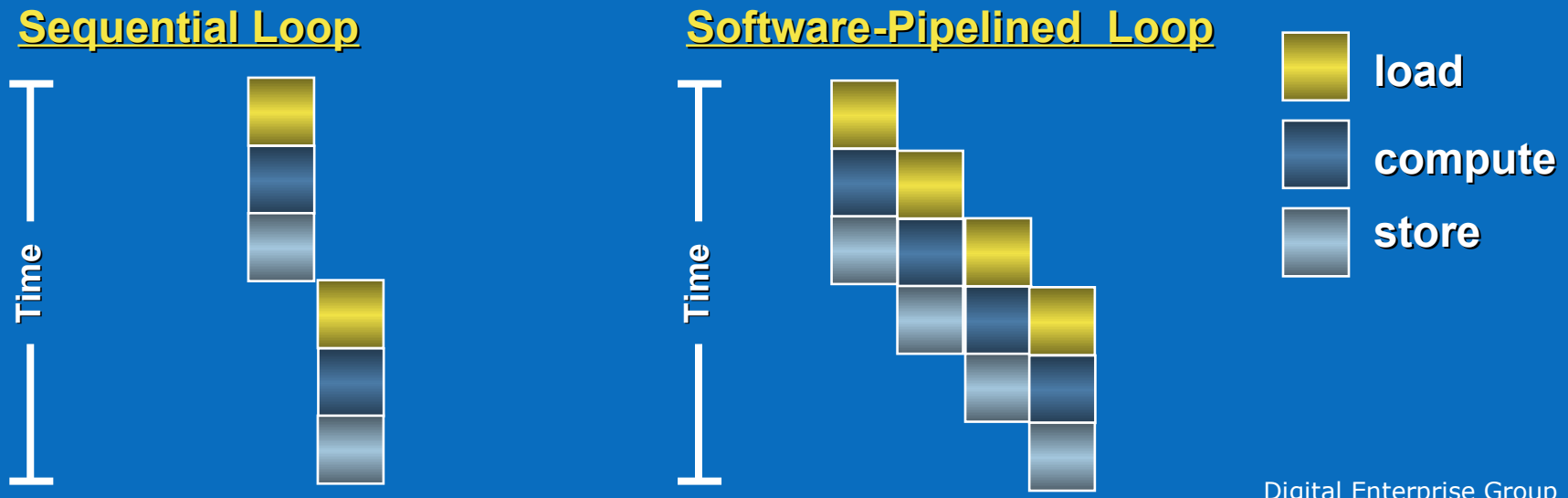
Software Pipelining

Traditional architectures use loop unrolling

- Results in code expansion and increased cache misses

The Intel® Itanium® Architecture provides application visible rotating registers

- Allows overlapping execution of multiple loop instances of the same code



Software Pipelining

- **Consider**

C code:

```
for (i = 0; i < n; i++)
        y[i] = a * x[i];
```

Pseudo Code:

```
loop:
        load xi
        fmul yi = a, xi
        store yi
        branch loop
```

- **Assume**
  - **Instruction latencies:**
    - **load** **4 cycles§**
    - **fmul** **2 cycles§**
    - **store** **1 cycle§**
    - **branch** **1 cycle§**
  - **Load, fmul, store and branch can be issued in the same instruction group.**

§Cycle counts for demonstration purposes only.

Software Pipelining

Cycle 1:	load x1		
Cycle 2:	load x2		
Cycle 3:	load x3		
Cycle 4:	load x4		
Cycle 5:	load x5	fmul y1=a,x1	
Cycle 6:	load x6	fmul y2=a,x2	
Cycle 7:	load x7	fmul y3=a,x3	store y1
Cycle 8:	load x8	fmul y4=a,x4	store y2
Cycle 9:		fmul y5=a,x5	store y3
Cycle 10:		fmul y6=a,x6	store y4
Cycle 11:		fmul y7=a,x7	store y5
Cycle 12:		fmul y8=a,x8	store y6
Cycle 13:			store y7
Cycle 14:			store y8

For n = 8

Prolog

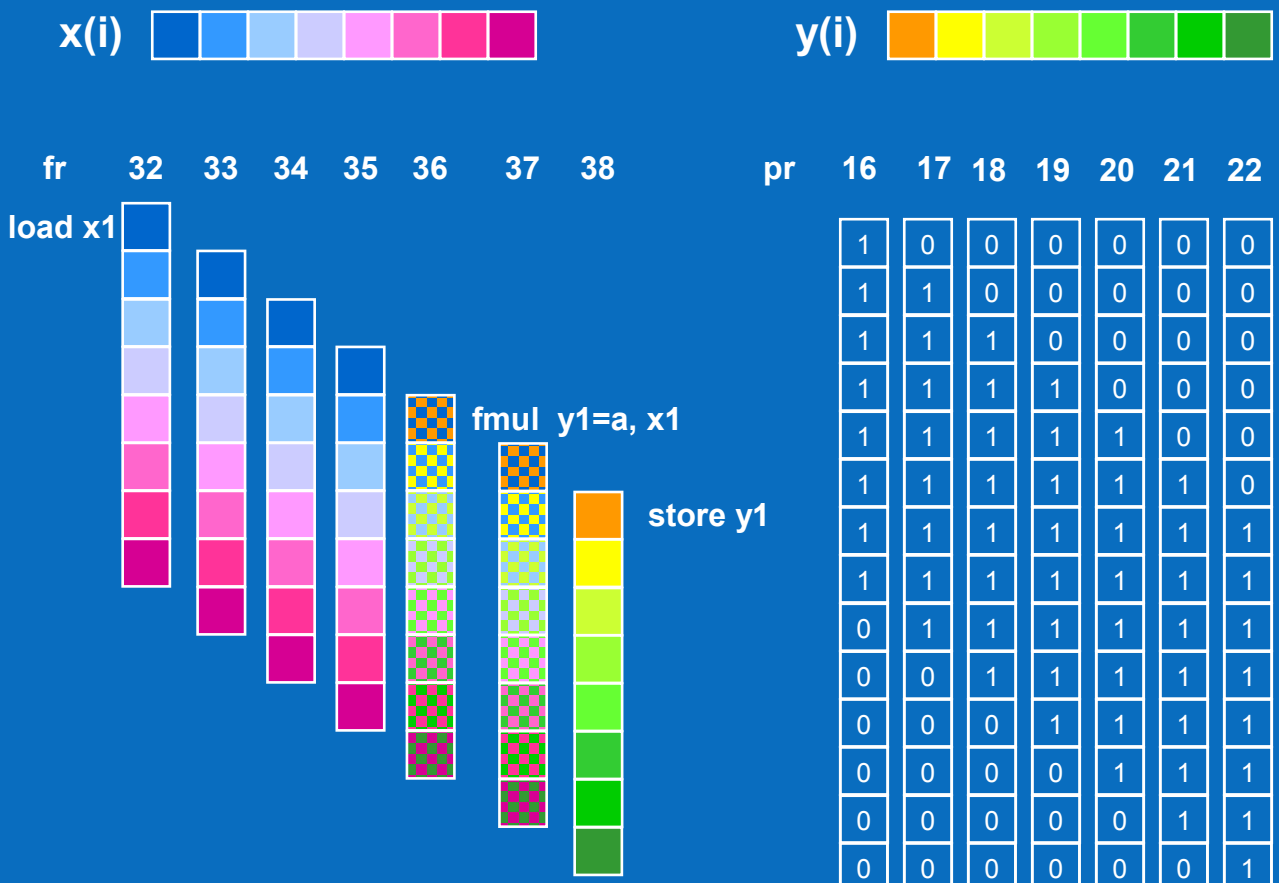
Kernel

Epilog

In this example, one iteration takes 7 cycles.

Cycle counts for demonstration purposes only.

Software Pipelining



Cycle counts for demonstration purposes only.

Software Pipelining

Actual code example:

```
    // Initialization
            mov pr.rot=0                    // Clear all rotating pred regs
            cmp.eq p16,p0=r0,r0             // Set p16=1
            mov ar.lc=7             // Set loop counter to n-1
            mov ar.ec=7                     // Set epilog counter # of stages
            ...
loop:
            { .mfi
(p16)       ldfd f32=[r32],8        // Stage 1: Load x
(p20)       fmpy.d f36=f6,f36       // Stage 5: y=a*x
            nop.i 0
            }
            { .mfb
(p22)       stfd [r33]=f38,8        // Stage 7: Store y
            nop.f 0
            br.ctop.sptk.few  loop          // Branch back
            }
```

Control Speculation

```
        cmp.lt  p1, p2 = r31, r2
(p2)    br.cond.spnt  exit
        ld8  r3 = [r4] ;;
        add  r6 = r3, r5
        ...
exit:   nop.b  0
```

Without speculated load and add

```
        ld8.s  r3 = [r4]  ;;
        ...
        add  r6 = r3, r5 ;;
        ...
        cmp.lt  p1, p2 = r31, r2
(p2)    br.cond.spnt  exit
        chk.s   r6, recv
next:   ...
exit:   nop.b  0

        // recovery code
recv:   ld8  r3 = [r4] ;;
        add  r6 = r3, r5
(p0)    br.cond.sptk  next
```

With speculated load and add

Use **chk.s** to test for deferred exception tokens.

Data Speculation

```
st4   [r2] = r31
ld4   r4 = [r3] ;;
add   r6 = r4,r5 ;;
sxt4 r7 = r6
```

Without advanced load

```
        ld4.a  r4 = [r3] ;;
        ...
        st4  [r2] = r31
        ld4.c.clr  r4 = [r3]
        add  r6 = r4,r5 ;;
        sxt4  r7 = r6
```

Advanced load with ld.c

```
        ld4.a  r4 = [r3] ;;
        ...
        add  r6 = r4,r5 ;;
        ...
        st4  [r2] = r31
        chk.a.clr  r4, recv
back:   sxt4  r7 = r6

recv:   ld4  r4 = [r3] ;;
        add  r6 = r4, r5
(p0)    br.cond.sptk back
```

Advanced load with chk.a

Use ***ld.c*** if only *load* is speculated, ***chk.a*** if *load* and its uses are speculated.

Hints

Cache management

- Explicit prefetch (lfetch, lfetch.fault)
- Temporal hints for loads, stores, and prefetch operations
  - .nt1 (not temporal in the first level cache)
  - .nt2 (not temporal in the first two cache levels)
  - .nta (not temporal in any cache level)
- Implementation and instruction specific behaviors
  - First cache for floating point is the second level cache
  - Current generation machines always allocate to L2 but may chose to bias allocated line for replacement

Branch prediction

- Taken vs not taken (.nt, .tk)
- Dynamic vs static (.sp, .dp)
- Implementation specific behaviors
  - Current generation of machines also use prediction hints to enable/control the pre-fetch engine

Floating Point

- Full IEEE support/Floating Point Register
  - Traps/faults under application control supported by EFI drivers
  - 82 Bit Floating Point Register
  - Instruction Set Supports:
    - Single, double, double-extended data types
- FMA - multiply-add instruction (f = a * b + c)
- Example:

```
fma.d f42=f8,f36,f41
```

- Arithmetic
  - Support for software divide and square root using reciprocal and reciprocal square root approximation
  - Max, min instructions for floating-point
- Data transfer
  - Load, store, GR <=> FR data transfer; load pair to double data

Memory Model

- Memory byte addressable using integer registers as address
  - Loads and stores should be properly aligned
  - Big and little endian supported natively
  - Special instructions to support 32-bit pointers
- Weakly ordered (not a bad thing)
  - Allows loads and stores completion to be re-ordered
  - Explicit ordering on semaphore operations or “volitle”
- Explicit ordering possible with acquire and release semantics
  - .rel ensures all previous memory operations are visible before the release is visible
  - .acq ensures all older memory operations are visible before the .acq is visible
  - mf has both .acq and .rel semantics
- Local ordering for same dataum access is ensured

Itanium® System Architecture

Enables adaptation to multiple Operating Systems

- Big and little endian support
- Multiple page sizes (4K-4G) with short or long format VHPT tables and pinned entries
- Multiple or single virtual address spaces
- Virtual memory regions with region specific properties
- Protection keys for safe OS/application data sharing and multiple privilege levels
- Low overhead interrupt and system calls
- Clean system architecture that simplify system interactions

Enables adaptation to multiple platforms

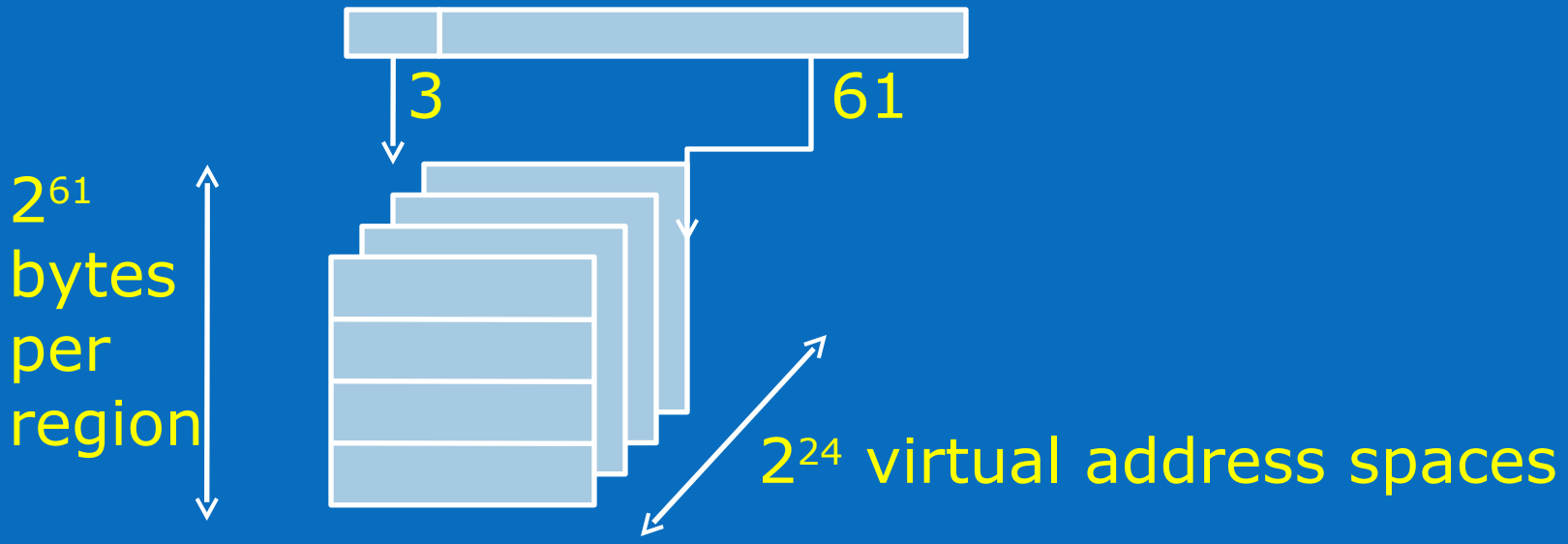
- Peta-bytes of addressability
- Efficient locking mechanisms for large symmetric systems
- Efficient memory coherency mechanisms

Provides flexible and secure building blocks to prevent unauthorized activities

Memory Management

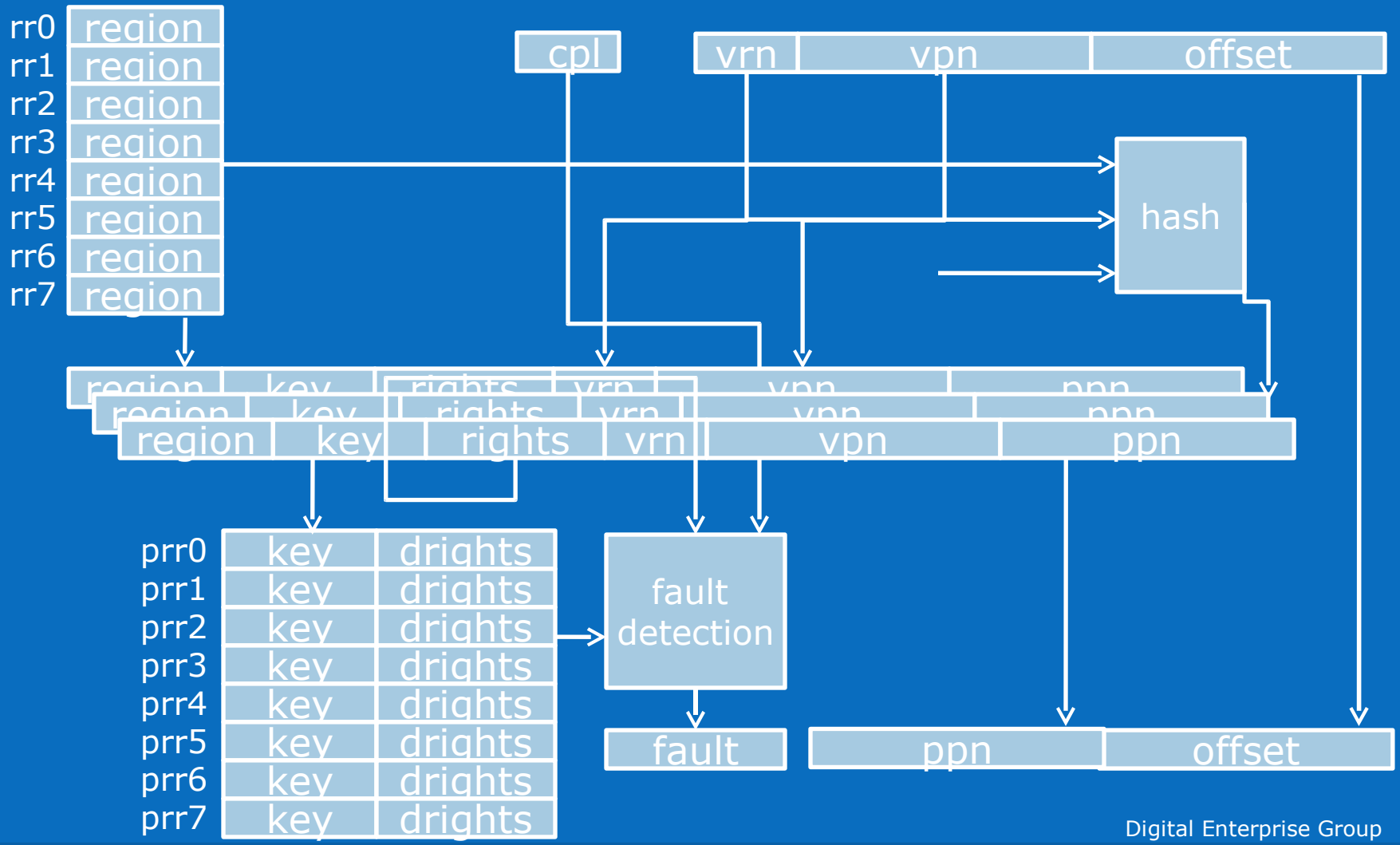
Supports both per process and global address space models

- Region identifier (VA[63:61]) accesses region registers
- Virtual Address = RR[VA[63:61]] VA[60:0]



- Page size support from 4K to 4G
  - 4K, 8K, 16K, 64K, 256K, 1M, 4M, 16M, 64M, 256M, 1G, 4G

Virtual Address Translation



Protection Architecture

Access granted based on current privilege level and access rights of the page

- 7 unique access right encodings; 4 unique privilege levels
- Promotion page only has RX and only available for access right=7
- Processor Status Register bits determine current privilege level, protection key enabling, and virtual memory enabling

Protection Keys

- Augmented security to facilitate a rich set of protection policies
  - Disable default access rights
- Protection key checking on all data, instruction, and RSE accesses
- Key miss and Permission faults allow OS to manage access to spaces
  - Enable/disable on a per process basis is possible

16 system registers serve as protection key cache

- Keys are 24 bits wide and each key has a set of RWX disable vectors

Learn More

Papers:

- IEEE Micro Mar/April 2003 –

  Itanium® 2 Processor Micro architecture

- IEEE Micro Mar/April 2005 –

  Montecito – A Dual-Core, Dual-Thread Itanium Processor

On-line:

http://developer.intel.com/design/itanium2/documentation.htm